DISTRIBUTED SIMULATION OF ASYNCHRONOUS COMPUTER ARCHITECTURES: THE PROGRAM DRIVEN APPROACH

G. Theodoropoulos, J. V. Woods
Department of Computer Science
University of Manchester
Oxford Road, Manchester M13 9PL, U.K.

ABSTRACT

Recently, there has been a resurgence of interest in asynchronous design techniques due to the potential of asynchronous logic for higher performance and lower power consumption. The research activity in this area has pointed to the need for suitable techniques for modeling and simulating asynchronous systems. Occam [1] allows the rapid development of distributed simulation models of asynchronous systems but its distributed nature introduces timing modeling problems. This paper presents an approach for dealing with these problems.

INTRODUCTION

Conventional synchronous architectures use design techniques based on global clocking whereby all the functional units operate in lockstep under the control of a central clock. As VLSI technology advances and systems become larger, faster and more complex, timing problems become increasingly severe and account for more and more of the design and debugging expense. Increased clock speeds make on-chip clock skew significant and interchip skew a major problem. One solution to clock-related timing problems is to use asynchronous design techniques without any global synchronization signals to control the rate at which different elements operate (Seitz 1980). Another potential advantage of asynchronous logic, which recently has led to a resurgence of interest in its use, is lower power consumption. An asynchronous system may be designed as a set of functional modules each operating at its own rate and cooperating through communication. The communication protocol synchronizes the modules involved in the communication and allows data to be shared between them. There exist many different approaches for designing asynchronous systems (Gopal. 1990). Sutherland's "Micropipelines" (Suther. 1989) use bundled data with an event-signaled handshake protocol for synchronization (figures 1,2). Following Sutherland's approach, the AMULET group at the University of Manchester have designed and implemented AMULET1, an asynchronous version of the ARM processor (Furber 1993).

The recent research activity in the area of asynchronous systems has pointed to the need for suitable techniques for modeling and simulating them (Udding 1991). Several notations and techniques have been suggested for this purpose (Hauck 1993). CSP in particular has attracted the interest of many researchers due to the strong relationship between its semantics and nondeterministic, concurrent behaviour of asynchronous systems (Martin 1986, Brunv. 1989).

[1] occam is a registered trademark of INMOS Group of Companies.

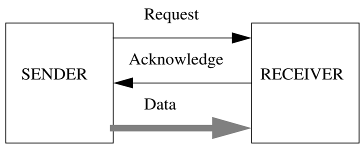

Figure 1: The bundled data interface of micropipelines

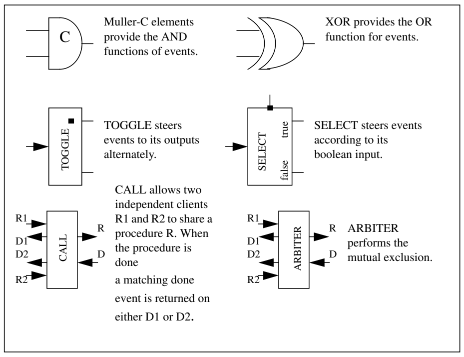

Figure 2: Event processing blocks

To support the design of AMULET1, a methodology for using occam, a CSP based parallel language that supports synchronous, unbuffered interprocess communication, for building Register Transfer Level executable models of asynchronous architectures has been developed (Theodor. 1994a) Using this methodology, an asynchronous architecture is modeled as a set of concurrent communicating occam processes. The processes are entirely data-driven, and self scheduled. Two channels are used for the synchronization of communicating processes, one for Request/data and one for the Acknowledge signal. To describe the nondeterministic behaviour of arbiters, the occam ALT statement is used.

TIMING ISSUES

The distributed nature of occam introduces a problem typical in distributed simulations, namely the problem of enforcing and maintaining strict temporal precision so that preemptions are avoided and causality constrains are not violated.

Since processes are data-driven, time is not required for synchronization and the correct operation of the occam simulation model (Theodor. 1994a). However time is still needed if the simulation model is to be used for a performance evaluation of the architecture; in order to get accurate performance results preemptions should not be allowed to occur.

In distributed simulations, preemptions occur if merge modules consume and process input messages from different channels in nonincreasing timestamp order. In a micropipelined architecture, micropipelines may be merged in one of the following ways:

- **Synchronous merge**. A functional module has to wait for all input data to become available before it starts its operation. This is the case when a **Muller-C** element is used for the corresponding request events. In the simulation model, the occam process has to wait for all input channels to "fire" and therefore no preemptions occur.
- **Data dependent merge**. The functionality of the system dictates the order in which messages from different source processes should be consumed and processed. This situation is implemented in hardware using a combination of a **select** and a **call** or **xor**. The process in this case behaves as a single input module, hence causality is not violated.
- **Arbitrated merge**. The order of arrival defines the order of consumption. If events from two micropipelines arrive at the same time, an arbitrary choice is made (arbiters); generally, this order will be different that the order in which the occam channels in the corresponding ALT fire and thus preemptions occur in this case.

The solutions that have been devised to deal with time modeling problems in distributed simulations fall into two categories, namely optimistic and conservative (Fujim. 1990). Conservative approaches will not let a process consume an event unless it is guaranteed that such a consumption will not cause any preemptions. This philosophy leads to deadlock situations. Three main techniques have been suggested to overcome this problem.

Deadlock Detection and Recovery techniques (Chandy 1981) allow the simulation to deadlock and then use a separate mechanism to detect if a deadlock situation has occurred and break it.

Synchronous approaches are based on global synchronization techniques in which periodically a global synchronization function is performed to enable the processes decide which events are safe to process.

Deadlock Avoidance methods (Chandy 1979) make use of Null messages to avoid deadlock situations. These techniques require that whenever a process finishes processing an event it sends a Null message to all its output links to inform its peers of its current clock value.

From the three aforementioned main conservative techniques, deadlock avoidance has the highest degree of compatibility with the modeling philosophy which exploits the relationship between the occam semantics and the behaviour of asynchronous architectures.

The Synchronous techniques require global synchronization which is neither easily implementable nor efficient on asynchronous distributed memory machines.

The Deadlock Detection and Recovery algorithms, like the optimistic approaches would increase the complexity of the occam simulation model forcing it to depart from the modeling philosophy described in previous sections.

The Deadlock Avoidance approach is quite simple to implement and does not require any modification of the structure of the simulation model; the model still maps the asynchronous architecture and no extra processes or links are needed except from some additional functionality to deal with Null messages.

Null messages are used only for synchronization purposes and do not carry any useful information. Their existence imposes a communication overhead in the network which may severely reduce the performance of the simulation. Asynchronous architectures in particular are communication bound systems and therefore the communication efficiency plays a crucial role in the performance of their simulation. To reduce the overhead due to Null messages, several variations of the Deadlock Avoidance technique have been suggested whereby Null messages are sent on a demand basis (Nicol 1984).

Asynchronous architectures are a special case of systems with certain characteristics and behaviour. The Program Driven Approach (PDA) proposed in this paper is a conservative deadlock avoidance technique which exploits this behaviour to achieve the minimization of the Null messages required. The following sections describe this approach.

The Program Driven Approach

Von Neumann computer architectures, synchronous or asynchronous, are deterministic systems: they accept as input instructions which they execute sequentially in a specific and predefined order. Each instruction defines the steps that are required for its execution as well as the behaviour of each functional module of the architecture. Consequently, the kind and sequence of events that occur in the system are determined at any time by the executing instructions. This ability to predict events in the architecture based on the information provided by the program under execution, forms the basis of the program driven approach; by looking at the instructions being executed the processes of the simulation model can decide whether an event is expected on a particular input link and thus whether their blocking upon this link would result to a deadlock. The key concept in the Program Driven philosophy is the "Instruction Lookahead Set":

Definition 1. *The* **Instruction Lookahead Set** *of a link λ is the set of instructions whose execution will potentially result to an event occurring on λ:*
$\mathbf{ILS}_\lambda = \{$*Instruction* $\mathbf{I}$: $\mathbf{I}$ *generates an event on link* $\lambda\}$.

As already mentioned, preemptions may occur only in arbiter processes and therefore it is only these that need to make decisions as to whether it is safe to accept an event. Based on the Instruction Lookahead Set defined above, the behaviour of arbiter processes with regard to message consumption may be specified as follows:

Rule 1. *An arbiter process Π is allowed to block and* *wait for an event on its input link λ during the execution of an instruction* $\mathbf{I}$ *if and only if* $\mathbf{I} \in \mathbf{ILS}_\lambda$.

The above rule ensures that arbiter processes block only for instructions that are likely to generate the corresponding events. However depending on the status of the system during the instruction execution such an event might not occur; in this case Null messages are required otherwise the arbiter process will become indefinitely blocked and the simulation model will deadlock:

Rule 2. *A process* Π *will produce a Null message on its output link* λ *while executing instruction* $\mathbf{I}$ *if and only if* λ *links process* Π *with an arbiter process,* $\mathbf{I} \in \mathbf{ILS}_\lambda$ *and for the current state of* Π, $\mathbf{I}$ *does not generate an event on* λ.

Clearly, the exploitation of the instruction lookahead results in the minimization of the Null messages and consequently the minimization of the overhead imposed on the communication system.

Such an exploitation presupposes that arbiter processes have knowledge as to which instructions are being executed. If this knowledge is not directly available, then an appropriate mechanism needs to be devised to provide arbiter processes with this information. Generally the functionality of the architecture being modeled will make the development of such a mechanism feasible. Otherwise the instruction lookahead properties of the system can not be exploited and rules 1 and 2 can not apply; in this case the conventional Deadlock Avoidance algorithm may be applied which would required a regular flow of Null messages on the input links of arbiter processes.

OCCARM: AN OCCAM SIMULATION MODEL OF THE ASYNCHRONOUS ARM

Using the approach described in the previous sections, OCCARM, a simulation model of the AMULET1 processor has been developed (Theodor. 1994b). OCCARM consists of a hierarchy of occam processes; it has the same degree of parallelism as the microprocessor and executes ARM machine code. Figure 3 shows the top level of the simulation model. The Address Interface operates autonomously and produces memory addresses starting from address 0; these addresses are synchronised within the Data Interface with output data from the execution pipe and are sent to memory. Input data from memory is directed to its appropriate destination depending on whether it is a data value (WrtCtrl) or an instruction (Decode1). The datapath control consists of three decode stages. Each stage includes a number of PLAs which generate the control signals required for the operation of the system; in the simulation model the PLAs are implemented as arrays of boolean values. Decode1 controls the operation of the Register Bank while Decode2 controls both the Shifter and the Multiplier. Decode3 controls the operation of the ALU; this is where the condition flags are checked, possibly changing the program flow. Invalid prefetched instructions following a branch are identified by comparing a parity flag in the ALU (the PCpar, changed every time a branch is taken) with the corresponding flag of the instruction. To make this mechanism more efficient an event is sent to Decode1 each time the parity changes so that instructions are discarded at the top of the pipeline. The Register Bank holds the 32 registers of the system and uses a novel

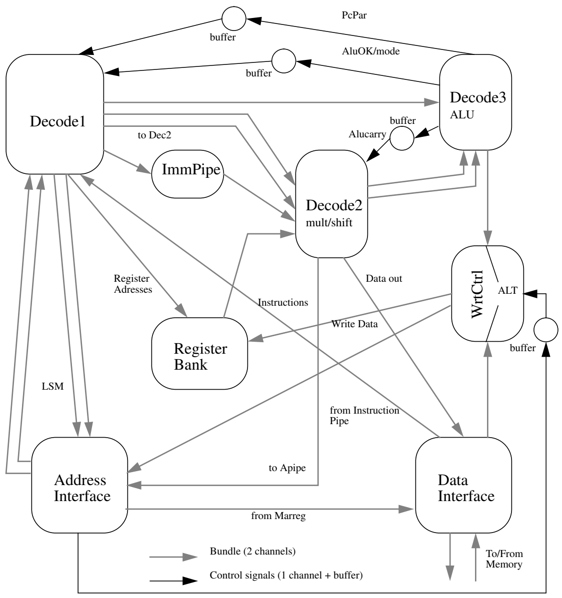

Figure 3: OCCARM top level

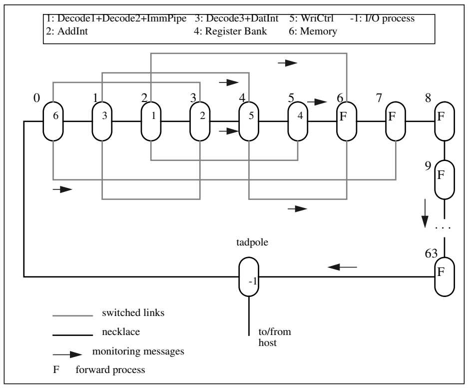

Figure 4: Mapping of the OCCARM onto the T-Rack

mechanism (the *lock fifo* (Paver 1994)) to deal with synchronization problems caused by the lack of a global clock.

The system used to host the simulation model is the ParSifal T-Rack, a reconfigurable 64-Transputer[2] machine, which has been developed at the University of Manchester. Two of the four links from each transputer of the T-Rack are permanently hardwired to form a processor chain known as the *necklace*. The off-necklace links may be connected by means of a crossbar switch which is built using twenty six INMOS C004 switch chips. Taking into account the restrictions imposed by the T-Rack and aiming at a) maximizing processor utilization and b) balancing communications, OCCARM has been distributed on the T-Rack as depicted in figure 4 yielding a speedup of 1.7 (Theodor. 1994a).

[2]Transputer is a registered trademark of INMOS Group of Companies

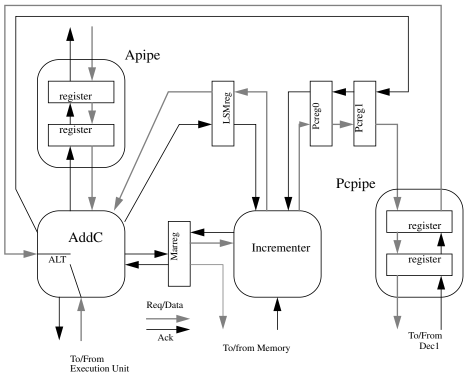

Figure 5: OCCARM: The Address Interface model

APPLYING PDA ON OCCARM

The Program Driven Approach has been applied to deal with preemptions in the OCCARM model. Two arbiters have been incorporated in AMULET1, in the Address Interface and Write Control units.

The Address Interface

The address Interface model is depicted in figure 5. One of the arbiter inputs is fed with PC addresses from the PC pipe. Thus in the simulation model there will be a continuous, instruction independent flow of events on the corresponding input link of the arbiter process(AddC) and therefore blocking on that link can not cause deadlocks. The other link receives messages from either the ALU or memory via the Write Control process. The **ILS** of this link is:

$\mathbf{ILS}_{AddC-WrtCtrl}$={*B, BL, SWI, LDR, STR, LDM, STM, Data Processing with PC as Dest. Reg.*}

AddC is an example of an arbiter process which has no direct knowledge regarding the executing instructions. An address produced by AddC is sent to Memory following the path: $AddC \Rightarrow MARegister \Rightarrow DataInt \Rightarrow Memory$. If it is an instruction address, the instruction message from memory is sent to Decode1 through the path: $Memory \Rightarrow DataInt \Rightarrow Decode1$. AddC is not in the path followed by the instruction and therefore has no information as to which instructions have entered the system; in order to apply PDA a mechanism needs to be devised to provide AddC with this information. Such a mechanism should be simple to implement and, more importantly, should not add extra communication overhead to the simulator.

Any attempt to use the second path above to inform AddC of the instruction currently in the path would involve an extra link connecting the path to AddC as well as extra messages. A neat and efficient solution is to take advantage of the hidden links in the above paths: the contra flow of the Acknowledge messages. Acknowledge messages are sent from register to register through control processes in the model. In the first path above, an address message produced by AddC will propagate to memory and from there to DatInt; DatInt will generate an Acknowledge message which will follow the opposite direction back to AddC. This Acknowledge message can be used to carry the corresponding instruction to AddC; no communication overhead is generated as the Acknowledge messages would be sent anyway.

The information required by AddC for making use of PDA is kept in a circular buffer, namely the "Instruction Lookahead Table" (*ILT*). Each instruction **I** received by the AddC is decoded and if $\mathbf{I} \in \mathbf{ILS}_{AddC-WrtCtrl}$ a new entry is appended in the *ILT*. Events arriving from PC pipe are let through for as long as *ILT* is empty; if *ILT* is non empty AddC blocks and waits for the corresponding message to arrive. There are two cases where an expected message will not arrive: a) If the condition codes of the instruction fail in the ALU; in this case a Null message is sent by the ALU to prevent deadlock, and b) If the instruction follows a branch; no Null messages are needed in this case since AddC gets informed whenever a branch is taken (a new address arriving from the ALU as a result of a branch will carry a new PCpar with it). Each time a branch is taken, the entries in the *ILT* corresponding to invalid prefetched addresses are removed. Any further instructions which subsequently arrive at the AddC and which are going to be discarded are just ignored.

The Write Control

Here, the application of PDA is straightforward. An event (data) will occur on *DatInt-WrtCtrl* link as a result of an event (data address) occurring on *Decode3-WrtCtrl* (see figure 3):

$\mathbf{ILS}_{DatInt-WrtCtrl}$={*LDR,LDM*}

WrtCtrl will continuously read messages arriving on *Decode3-WrtCtrl*; if it encounters a data address message, then it blocks and wait for the corresponding data on the *DatInt-WrtCtrl* link. This message will always arrive so there is no need for Null messages. Similarly to AddC, an Instruction Lookahead Table is maintained by WrtCtrl.

IMPROVING PERFORMANCE: THE LATENCY LOOKAHEAD

The application of PDA on OCCARM has resulted to a 20% decrease of the performance of the simulator achieved when arbiters are modeled by ALT statements and no attempt is made to prevent preemptions (which is the maximum possible performance): $Performance_{PDA} = 0.8 Performance_{ALT}$.

This figure can be improved by allowing PDA to exploit the concurrency of the system. As soon as it is informed that an instruction being executed is in the $\mathbf{ILS}_{\lambda_1}$ of one of its input links λ_1, an arbiter process will stop accepting any messages arriving on its other input link λ_2 until the expected event on λ_1 occurs. As a consequence of this behaviour, all the processes that are part of the path that leads to λ_2 will block and wait, and the pipelines at the output side of the arbiter process will starve. Thus large parts of the simulator will remain idle for substantial periods of time.

A solution to this problem is to provide arbiter processes with some indication as to when in the simulated future a message they expect will actually arrive. This information would enable them to consume a number of events occurring on λ_2 link before they block on λ_1 increasing thus the concurrency of the simulation model.

This information can be obtained by taking into account the propagation delays of the pipeline stages in the architecture. An event generated by an instruction will propagate through a number of pipeline stages before it reaches an arbiter. The path followed by the event is completely defined by its parent instruction; the latency of the path however is not deterministic but depends on the number of elements in the micropipelines of the path at any particular time. Thus it is impossible to know in advance the exact time required for an event to propagate down a path. This is due to the asynchronous, delay insensitive nature of the architecture. However, there is a lower bound to this time, namely the latency of the path when its micropipelines are empty.

Based on this information, the prediction regarding the occurrence of the event on a link is straightforward:

Definition 2. *The* **The Minimum Latency Lookahead** *of a link λ during the execution of instruction I* $\mathbf{MLL}_{\lambda,I}$*, is defined as the total propagation delay of the path leading to λ, when the pipelines of the path are empty:*
$\mathbf{MLL}_{\lambda,I} = \sum Delay_i, Delay_i$ *= Propagation Delay of the i-th Pipeline Stage in the Path* [3]

The path may be chosen so that it makes the calculation of the **MLL** easier. For example, in the case of OCCARM, for the calculation of $\mathbf{MLL}_{WrtCtrl-AddC,I}$, the "$Memory \Rightarrow DatInt \Rightarrow Decode1 \Rightarrow Decode2 \Rightarrow Decode3 \Rightarrow WrtCtrl \Rightarrow AddC$" path is considered, while for the calculation of $\mathbf{MLL}_{DatInt-WrtCtrl,I}$ "$WrtCtrl \Rightarrow DatInt \Rightarrow Memory \Rightarrow DatInt \Rightarrow WrtCtrl$" is taken into account.

Preliminary experiments with OCCARM have indicated that by exploiting the Latency Lookahead of AMULET1 a performance improvement of 10% can be achieved.

CONCLUDING REMARKS

This paper has described a technique for maintaining temporal precision in distributed simulation models of asynchronous computer architectures, namely the Program Driven Approach. This is a conservative, deadlock avoidance approach that exploits the characteristics of asynchronous architectures to achieve efficiency and high performance. It is *program driven* in the sense that the behaviour of the processes in the model with regard to message consumption is determined by the instructions that are executed in the system at any particular moment.

The application of this approach depends on the exploitation of the instruction lookahead properties of the system which might not always be possible or efficient. Whenever the characteristics of the architecture permit such an exploitation however, PDA will provide the theoretical framework for the development of simple, efficient and fast deadlock avoidance algorithms.

References

Brunvand, E.; and R.Sproull. 1989. "Translating Concurrent Communicating Programs into delay-Insensitive Circuits", Technical Report CMU-CS-89-126, Department of Computer Science, Carnegie Mellon University (Apr.).

Chandy, K. M.; and J.Misra. 1979. "Distributed Simulation: A Case Study in the Design and Verification of Distributed Programs", IEEE Trans. on Soft. Eng., SE-5, no. 5 (May): 440-452.

Chandy, K.; and J.Misra. 1981."Asynchronous Distributed Simulation via a Sequence of Parallel Computations", Communications of the ACM 24, no. 4 (Apr.): 198-206.

Fujimoto, R. 1990. "Parallel Discrete Event Simulation", Communications of the ACM 33, no. 10, (Oct.): 31-53.

Furber, S.; P.Day; J.Garside; N.C.Paver; and J.V.Woods. 1994. "A Micropipelined ARM", In Proceedings of the VLSI'93 Conference, (Grenoble, France, Sept. 6-10), 5.4.1-5.4.10.

Gopalakrishnan, G.; and P.Jain. 1990. "Some Recent Asynchronous System Design Methodologies". Technical Report UU-CS-TR-90-016. Department of Computer Science, University of Utah (Oct.).

Hauck, S. 1993. "Asynchronous Design Methodologies: An Overview". Technical Report UW-CSE-93-05-07. Department of Computer Science, University of Washington (Apr.).

Jefferson, D. 1985. "Fast Concurrent Simulation using the Time Warp Mechanism", In Proceedings of the 1985 Conference on Distributed Simulation , Society for Computer Simulation (Jan).

Martin, A.J. 1986. "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits", Distributed Computing 1, no. 4 (Apr.): 226-234.

Nicol, D. M.; and P.F.Reynolds.1984. "Problem Oriented Protocol Design", In Proceedings of 1984 Winter Simulation Conference, (Dec.), 471-474.

Paver, N.C. 1994. "The Design and Implementation of an Asynchronous Microprocessor", PhD Thesis, Department of Computer Science, University of Manchester.

Seitz, C.L. 1980. "System Timing". In Introduction to VLSI Circuits, C.Mead and L.Conway, eds. Addison Wesley, chapter 7.

Sutherland, I. 1989. "Micropipelines", Communications of the ACM 32, no. 6 (Jun.): 720-738.

Theodoropoulos, G; J.V. Woods. 1994. "Building Parallel Distributed Models for Asynchronous Computer Architectures", to be presented at the World Transputer Congress 1994, (Italy, Sept. 5-7).

Theodoropoulos, G. 1994. "An occam model of the AMULET1", In Proceeding of the AMULET Modeling Workshop, (Windermere, Cumbria, England, Jul. 18-22).

Udding, J. T. 1991. "Formal Models", In Proceedings of the Workshop on the Design and Implementation of Asynchronous Circuits, (Amsterdam, Nov.), 12-16.

BIOGRAPHY

George Theodoropoulos received the Diploma degree in Computer Engineering from the University of Patras, Greece, in 1989 and the M.Sc degree in Computer Science from the University of Manchester, U.K., in 1991. He is currently working towards a PhD in Computer Science with the AMULET group at the University of Manchester, U.K. His research interests are in the areas of parallel processing, asynchronous systems and networks.

[3] The inclusion of **I** in the definition indicates that the **MLL** is instruction dependent.